

# Software Engineering

CS305, Autumn 2020

Week 4

# Class Progress...

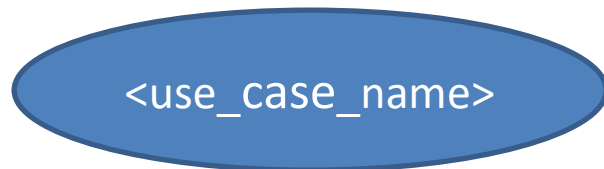
- Last week:
  - Requirements Engineering Detailed Steps
    - Elicit, Analyze, Specify, Validate, Manage change
  - Requirements modeling
    - Goal-oriented, text-based methods, graphical based methods
  - Object Oriented Analysis and Design – overview
    - Object Modeling Technique
    - Unified Modeling Language (UML) and structural diagrams

# Class Progress...

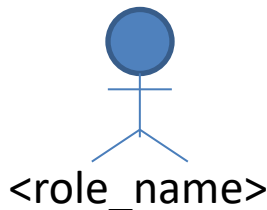
- This class: UML behavioral diagrams
  - Describe behavior or dynamic aspect of the system
  - E.g.
    - Use Case diagram / user stories / scenarios

# Use Case Diagrams

- Describes outside view of the system
  - Interaction of *outside entities* (Actors) with the system
  - System actions that result in observable actions of value to the actors
- Notation (important ones):



Use case



Actor



Connector between actor and use case  
(indicates “is the actor of”)

# Actor

- Entity: human or device that interacts with the system
- Plays some role
  - Can play more than one role
    - E.g. customer of a bank can also be an employee of the bank (customer and employee are roles)
  - More than one entity can play the same role
    - E.g. an employee and a regular customer can both play the role of a *customer*
  - Can appear in more than one use case

# Running Example

1. Registrar sets up the curriculum for a semester using a scheduling algorithm
2. One course may have multiple course offerings (think: sections)
3. Each course offering has a number, location, and time
4. Students register for courses using a registration form
5. Students may add/drop courses for a certain period after registration
6. Professors use the system to receive their course attendance sheets / course rosters
7. Users of the system are assigned passwords to validate at logon

# Exercise: Identify Actors

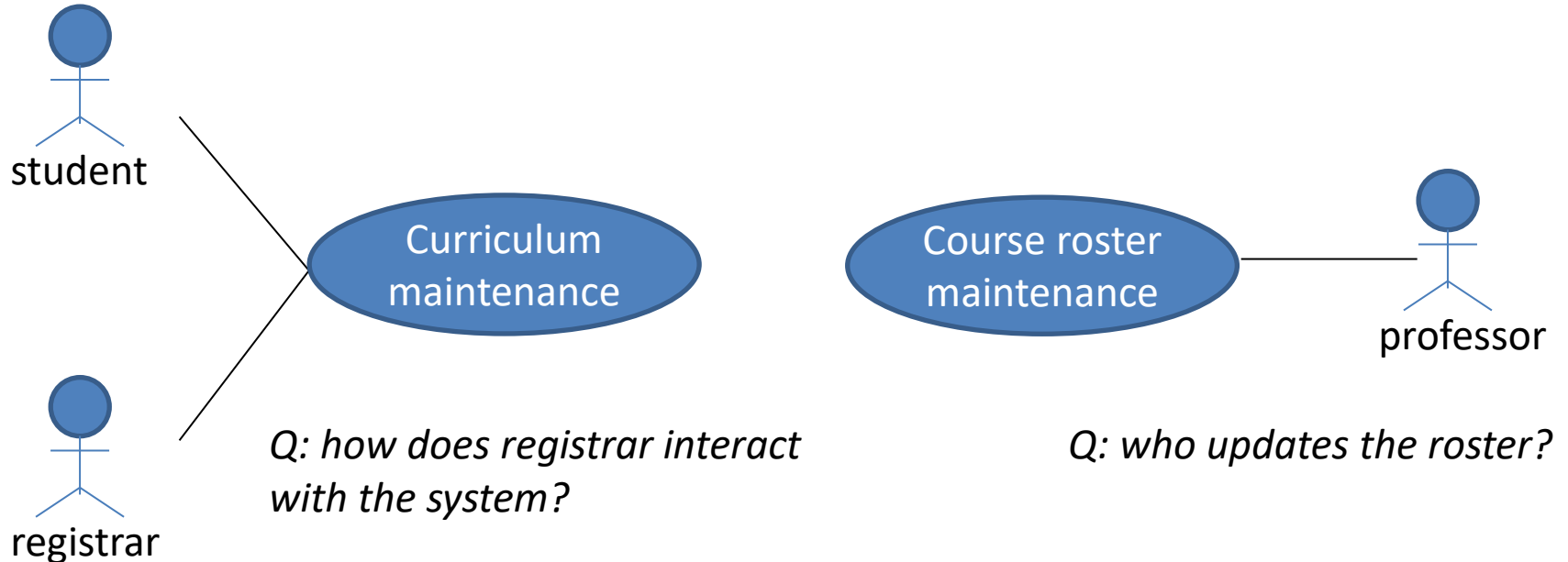
1. Registrar
2. curriculum
3. Semester
4. Scheduling algorithm
5. Course
6. Course offerings
7. Students
8. Registration form
9. Professors
10. Passwords

# Exercise: Identify Actors

1. Registrar
2. Curriculum
3. Semester
4. Scheduling algorithm
5. Course
6. Course offerings
7. Students
8. Registration form
9. Professors
10. Passwords



# Example Use Case Diagrams



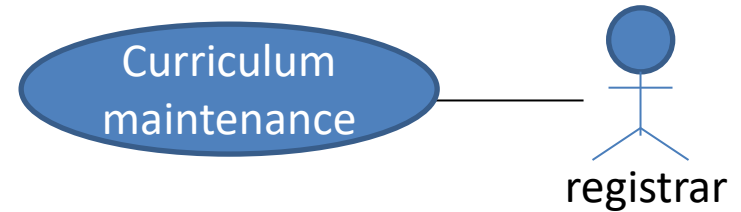
*Q: how to document the interactions?*

# Documenting use case - guidelines

- Describe flow of events either *formally or informally*
  - How the use case starts and ends
  - Normal flow of events
  - Alternative flow of events
  - Exceptional flow of events
- Formal way
  - Sequence diagrams, pseudocodes
- Informal way
  - Textual description

# Documenting use case - example

- Registrar provides a password to log in to the system
- If the password is valid, the system asks to specify a semester
- Registrar enters the desired semester, and the system prompts the registrar to select an activity: ADD / DELETE / REVIEW / QUIT
- When selected ADD / DELETE, the system allows registrar to add / delete a course
  - When done, the system runs the scheduling algorithm
- When selected REVIEW, the system displays the curriculum for that semester
- When selected QUIT, the system logs out the registrar



# Use cases - role

- Why important?
  - More effective requirements elicitation
  - Starting point for analyzing architecture (next topic)
  - Identify priority of users (e.g. Registrar. *If the registrar cannot perform his assigned role? How can a student use the system?*)
    - Help in better planning
  - Help in writing test cases even before the system is defined / coded

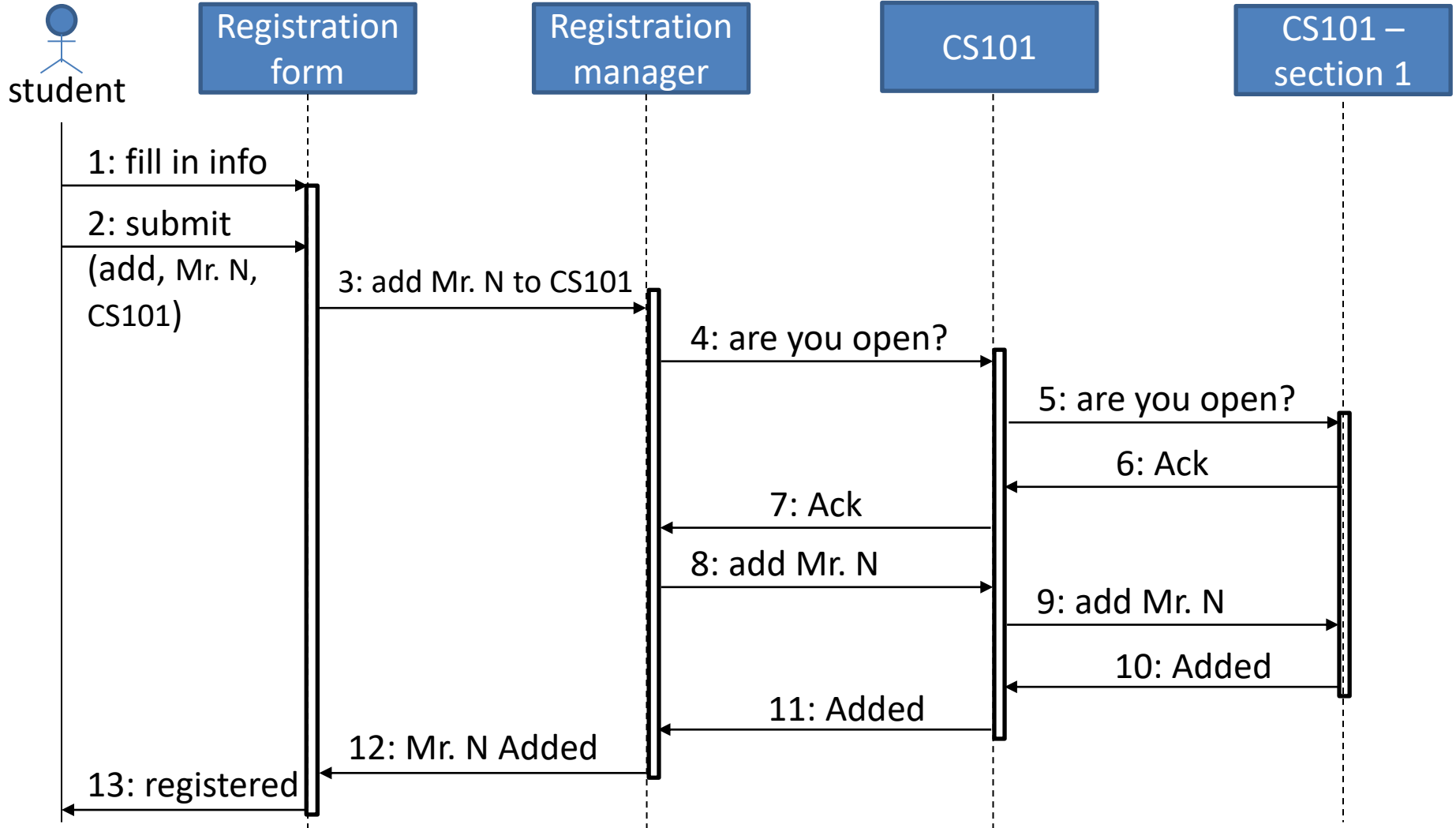
# Use case diagram creation - guidelines

- Choose a name that conveys purpose
- Put a single scenario into a use case
- Define the flow of events clearly - helps understand how system works
- Omit irrelevant details
- Extract common flow of events among multiple user interactions to create new use cases i.e. refine e.g. Registrar, student, professors all log in to the system before performing their roles.

# Sequence Diagrams

- Interaction diagram that describes how objects / components communicate and the ordered sequence of messages that are exchanged
- Can be used as a formal way to document a use case

# Sequence Diagrams - Example



# Sequence diagram creation - guidelines

- Draw **objects** that participate in the interaction at the top along X-axis
  - Place objects that *initiate the interaction* towards the left
- Add object **lifelines** – lines that show the existence of an object over a period
  - Add dashed lines for all except the left-most object
- Place **messages** from top to bottom
  - Annotate messages with numbers for added clarity
- Add **focus of control** – thin rectangular boxes that indicate the period when the object is in action

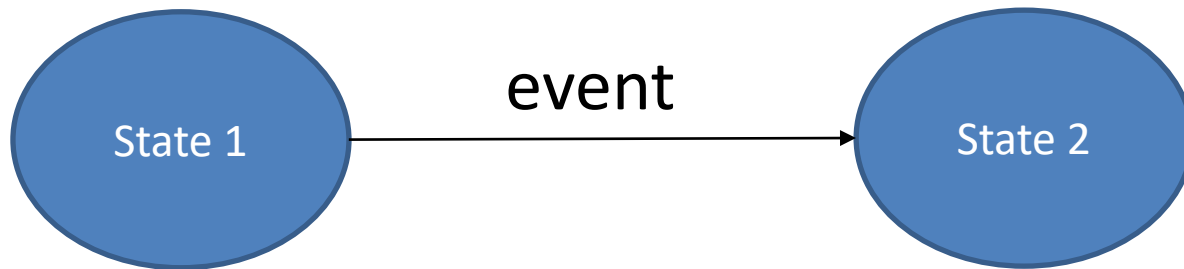


# State Transition Diagrams

- Shows possible life history of each object / class
- Defined for each relevant object / class
- Shows:
  - States of the class (attributes)
  - Events that cause transition from one state to another
  - Actions that result from state change

# State Transition Diagrams - Notation

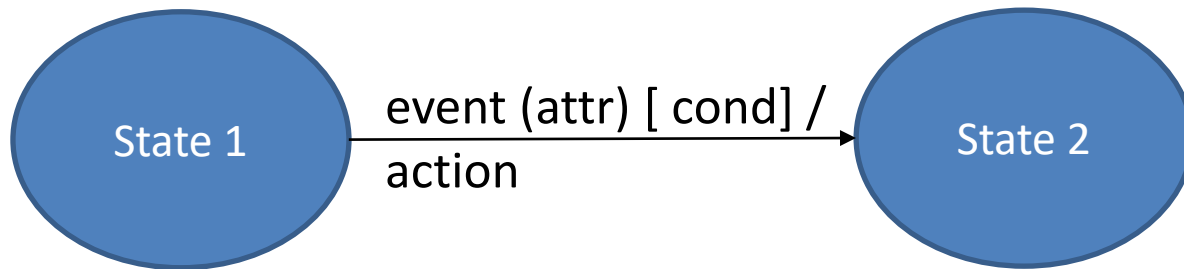
- States are indicated as ovals with names written inside



- Transition is indicated as event that triggers the transition. Indicates passage from one state to another because of some external stimuli (some events may be consumed within the state itself)

# State Transition Diagrams - Notation

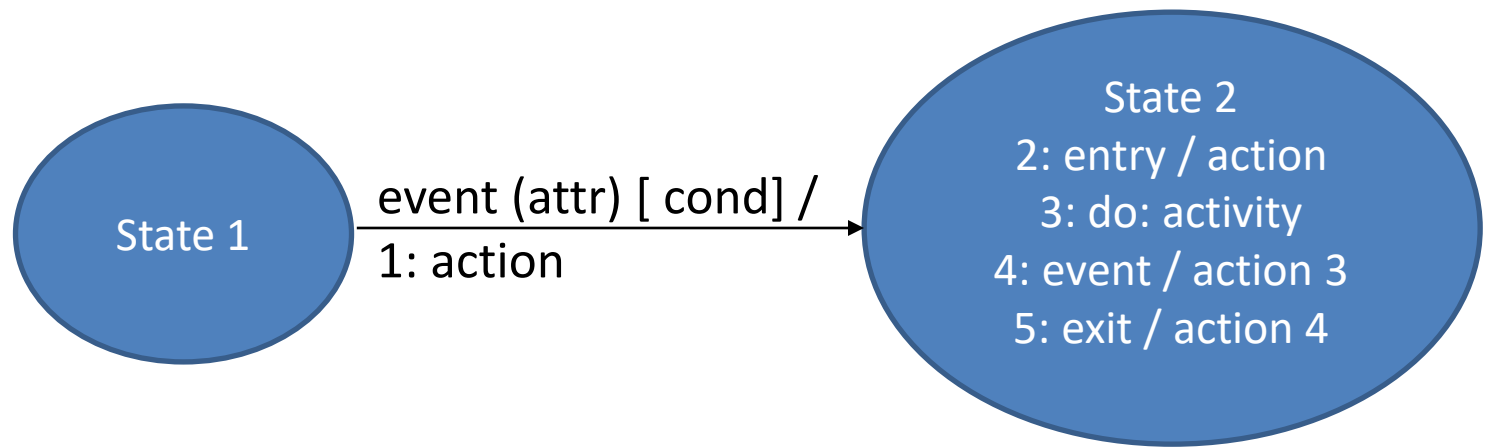
- events might also produce actions



- might also have attributes (analogous to method parameters) and Boolean conditions that indicate that the event is triggered only when the condition is true

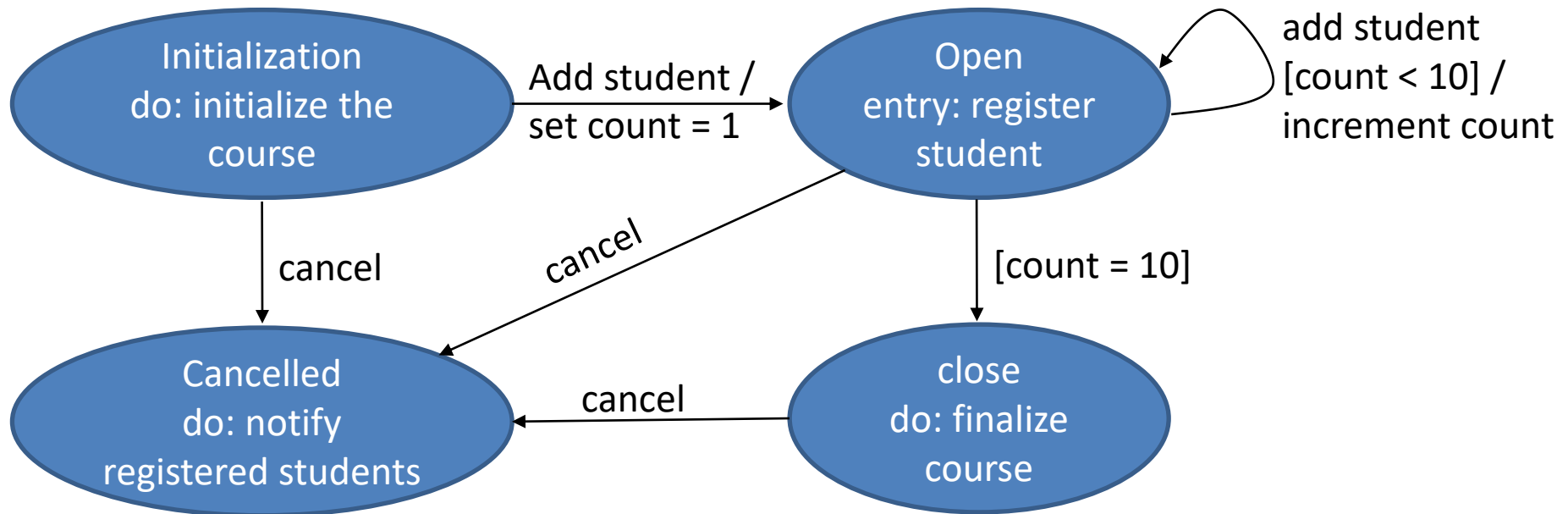
# State Transition Diagrams - Notation

- States might also be associated with activities and actions

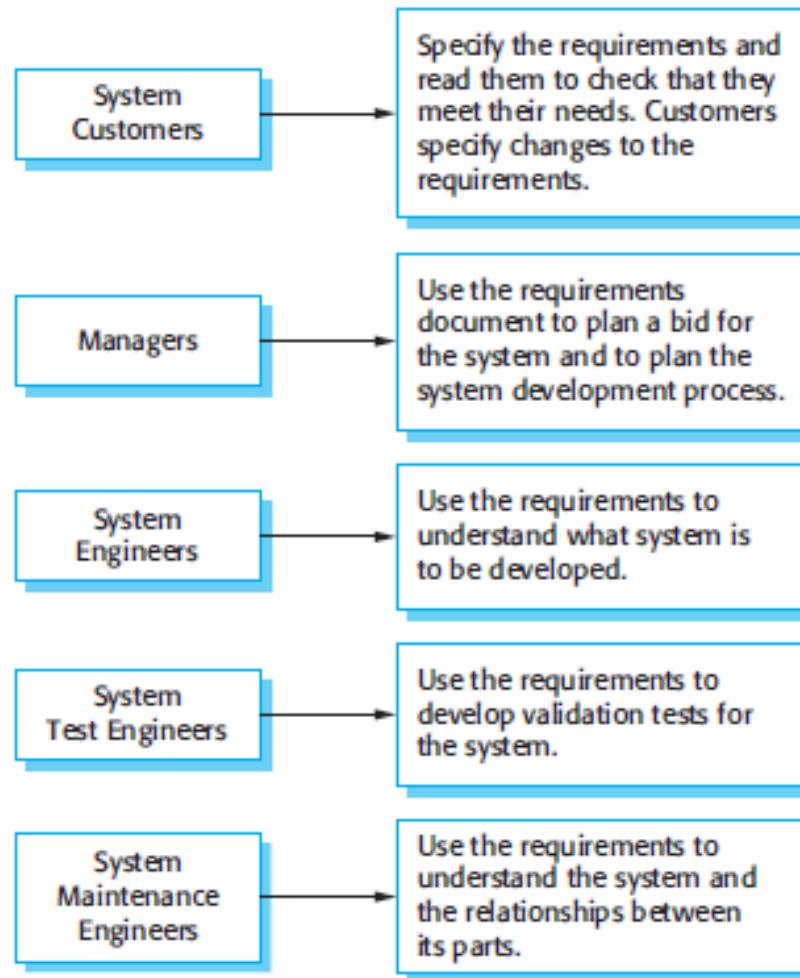


- activities: operations performed by the object in a given state that take time to complete
- actions: events that can be triggered upon entry or exit to that state or in response to specific event caused due to an activity performed
- Numbers indicate the time ordering of actions / activities

# State Transition Diagrams - example



# Users of a Requirements Document



# SRS Summary

- Way to communicate requirements to others
- Different projects require different SRSs depending upon the context e.g. small vs. large teams

# Time to turn things around... a bit.

*Write tests before you code and then code to make the tests pass*



# Testing

- Is a kind of verification technique
  - Recall: verification is checking against requirements
- Is executing the program on a *tiny sample* of the input domain
  - It is a dynamic technique: you need to execute the program
  - It is an approximation technique: for all other inputs, you expect the behavior of the program to be consistent with the samples tested

# Testing

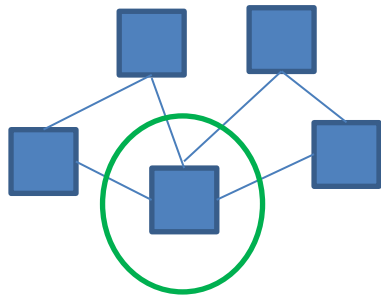
- Goal is to uncover bugs in the program

*“A test is successful if the program fails” –*

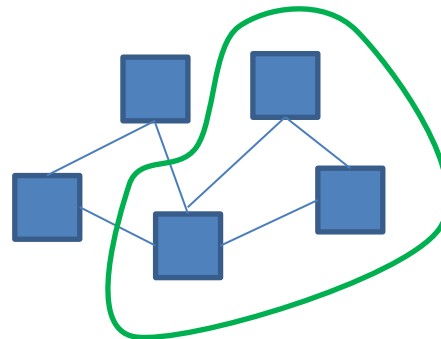
*Goodenough and Gerhart*

# Testing Granularity Levels

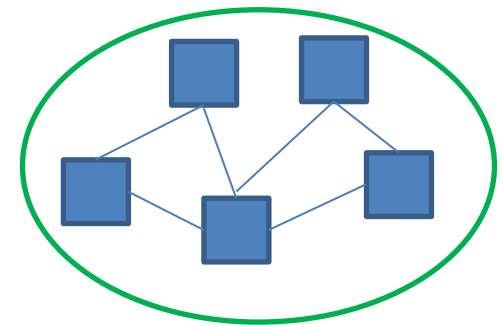
- View: software system as a bunch of interacting components



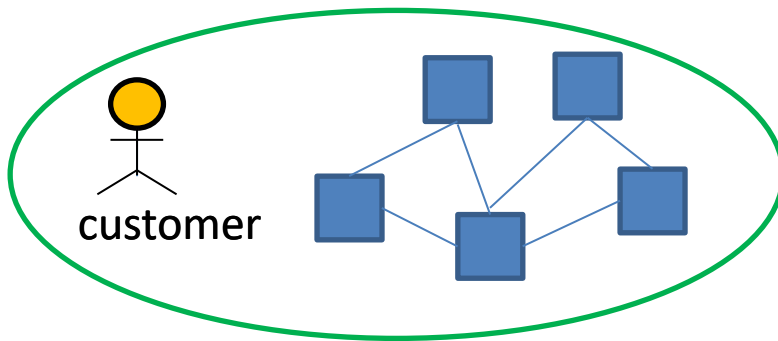
**Unit testing**



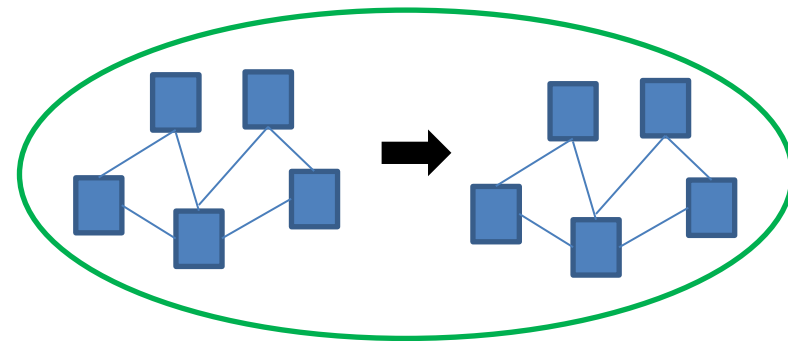
**Integration testing**



**System testing**



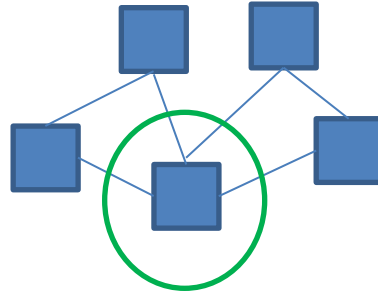
**Acceptance testing**



**Regression testing**

# Testing Granularity Levels - Overview

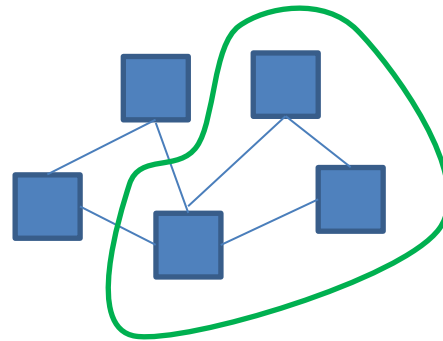
- Unit Testing



- Testing of individual modules in isolation

# Testing Granularity Levels - Overview

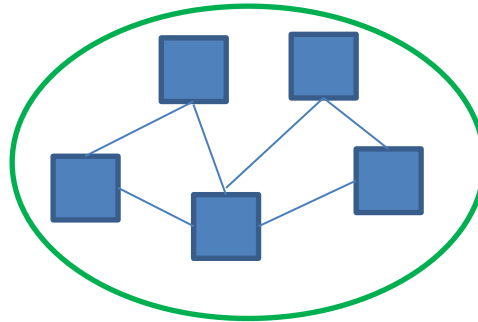
- Integration Testing



- Testing of a subset of modules taken together
  - Testing for interaction among the modules
  - Modules of the subset can be tested one at a time or all taken together (Big-bang)

# Testing Granularity Levels - Overview

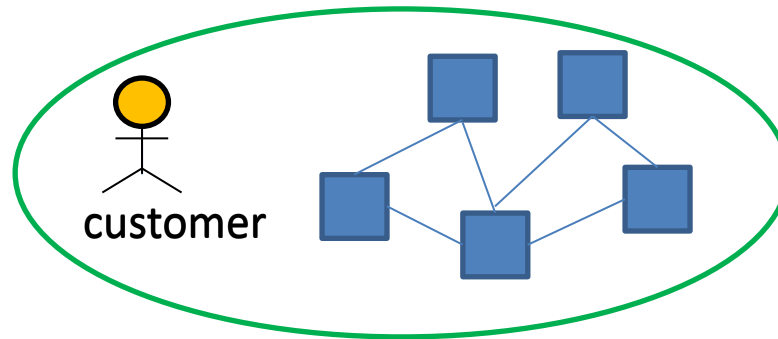
- System Testing



- Testing complete system as a whole: functional and non-functional requirements
  - Functional tests: test the functionality provided by the system
  - Non-functional tests: assess the “..ility” of the system – usability, reliability, maintainability etc.. e.g. load and stress tests

# Testing Granularity Levels - Overview

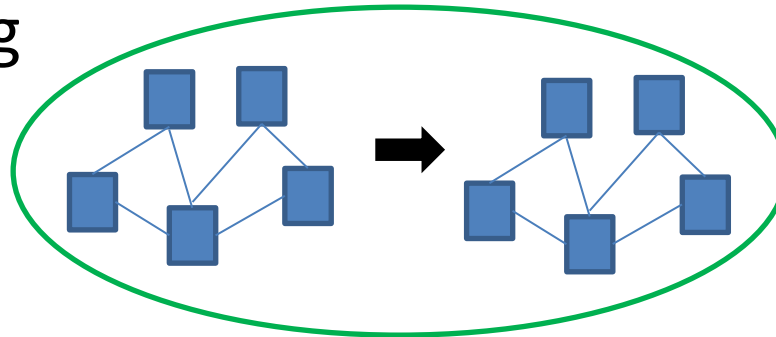
- Acceptance Testing



- Testing complete system as a whole: validation of software against customer requirements
  - System does what the customer expects it to do

# Testing Granularity Levels - Overview

- Regression Testing

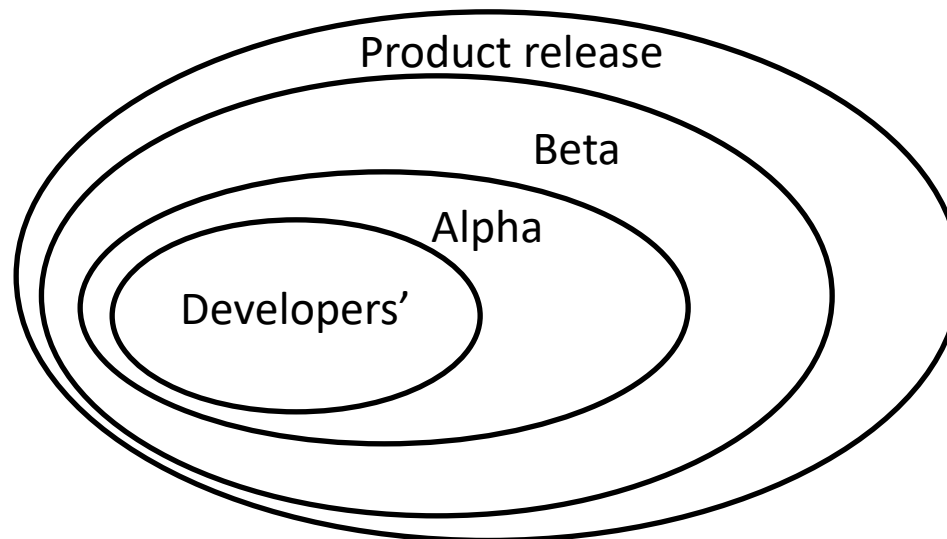


- Testing complete system as a whole: tests that check if some changes negatively affect the parts that have not changed
  - One of the causes why software maintenance is so expensive
  - Automation is an active research focus area



# Alpha and Beta Testing

- Alpha Testing
  - Release the software to users within the organization for testing
  - Tolerance to bugs is fairly high
- Beta Testing
  - Release the software to a selected list of users outside org.



# Black-box and White-box Testing

- Two families of test strategies
- Black-box testing: based on functionality
  - Do not look inside i.e. the code
  - Test against software description
  - Cannot reveal errors due to incorrect implementation
- White-box testing: based on code
  - test all control paths: sequence of code lines
  - Cannot reveal errors due to missing paths i.e. missing functionality


# Black-box Testing Example

- Specification: input an integer and print it

```
1. void printNumBytes(int param) {  
2.     if ( param < 1024 )  
3.         printf(“%d”,param);  
4.     else  
5.         printf(“%d Kb”,param/124)  
6. }
```

- The implementation details are a grey-area
  - Cons: miss testing inputs that are > 1024
  - Pros: need not know the internal functionality to test

White-box testing  
would catch this typo.



# White-box Testing Example

- Note: test without a specification

```
1. int fun(int param) {  
2.     int result;  
3.     result = param / 2;  
4.     return result;  
5. }
```

- Execute all statements in the function
- Cons: miss catching an obvious error for a specification: input an integer and return half the value if even. Unchanged otherwise.

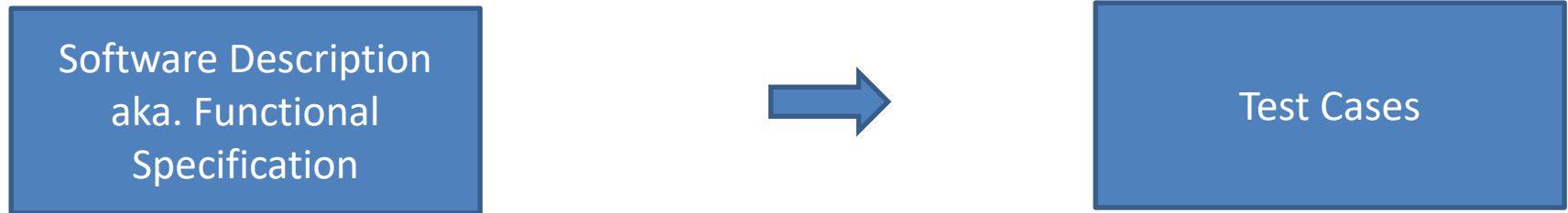
# Focus: Black-box Testing

# Black-box Testing

- Advantages:
  - Focuses on the domain
  - Does not need code. Helps you start early. Real advantage in real-life software development
  - Catches logic errors
  - Applicable at all granularity levels

*We will focus on system testing at this point*

# Black-box Testing – From Spec. to Test Cases



- **Input:** Function Spec.
- **Output:** Test Cases – set of inputs and corresponding outputs that we use to exercise our code to uncover bugs
- *Problem: How do we go from input to output?*
  - *Can be extremely complex*
- **Solution:** break the complexity
  - 4 main steps

# From Spec. to Test Cases

- **Step 1:** identify independently testable features



- May not be possible to test all possible features at once
  - e.g. printing receipt after successful ATM transaction



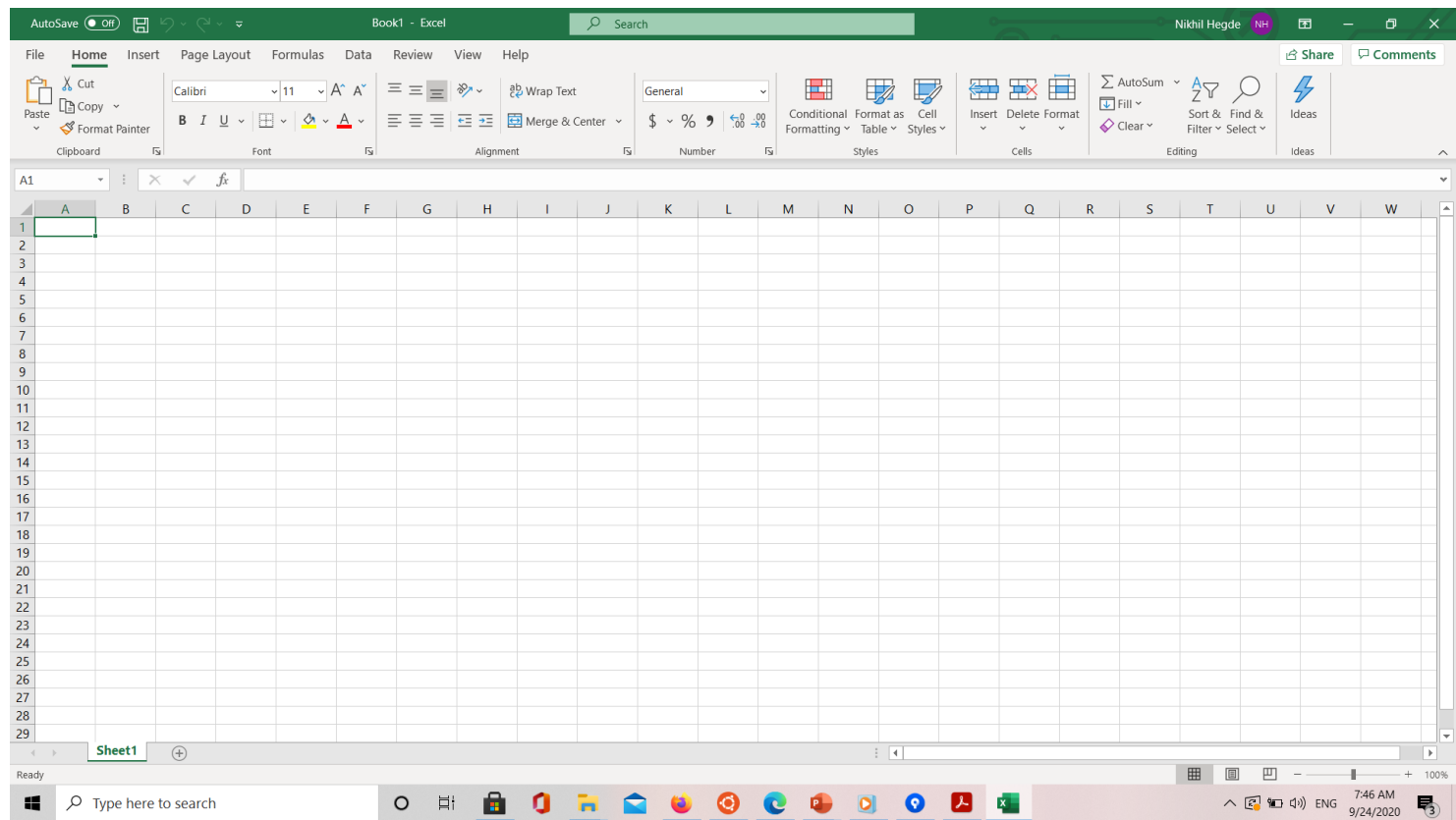
# Identifying Independent Features

- How many features here (sum)?

```
1. int sum(int a, int b) {  
2.     return a + b;  
3. }
```

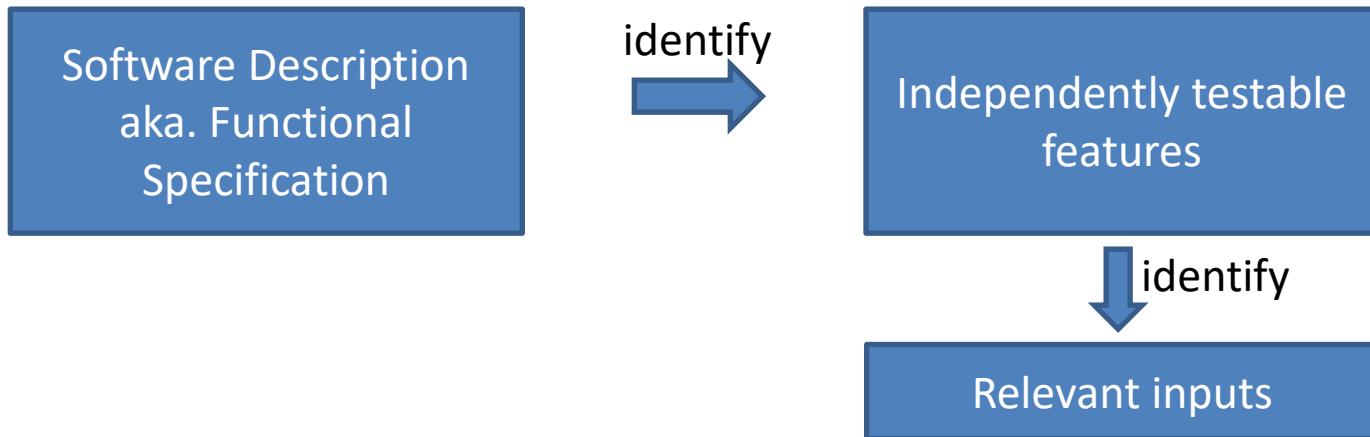
# Identifying Independent Features

- How many features here (MS-Excel)?

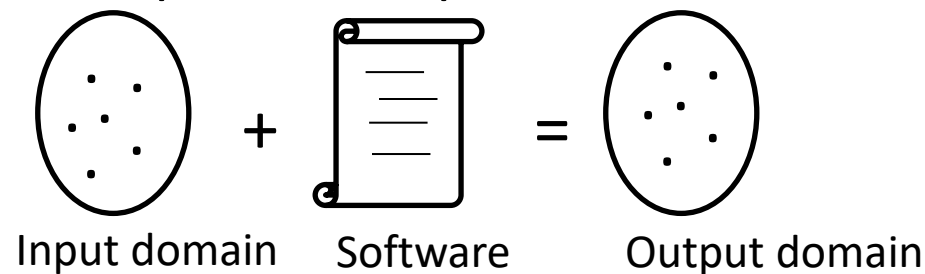


# From Spec. to Test Cases

- **Step 2:** identify relevant inputs. Also called **test data selection**



- test cases = inputs + expected outputs



# Test Data Selection – Naïve approach

- Test-them all !
  - Consider all inputs from the input domain (exhaustive testing)
  - Lay-man approach
- E.g. How long would it take to test the sum function exhaustively:

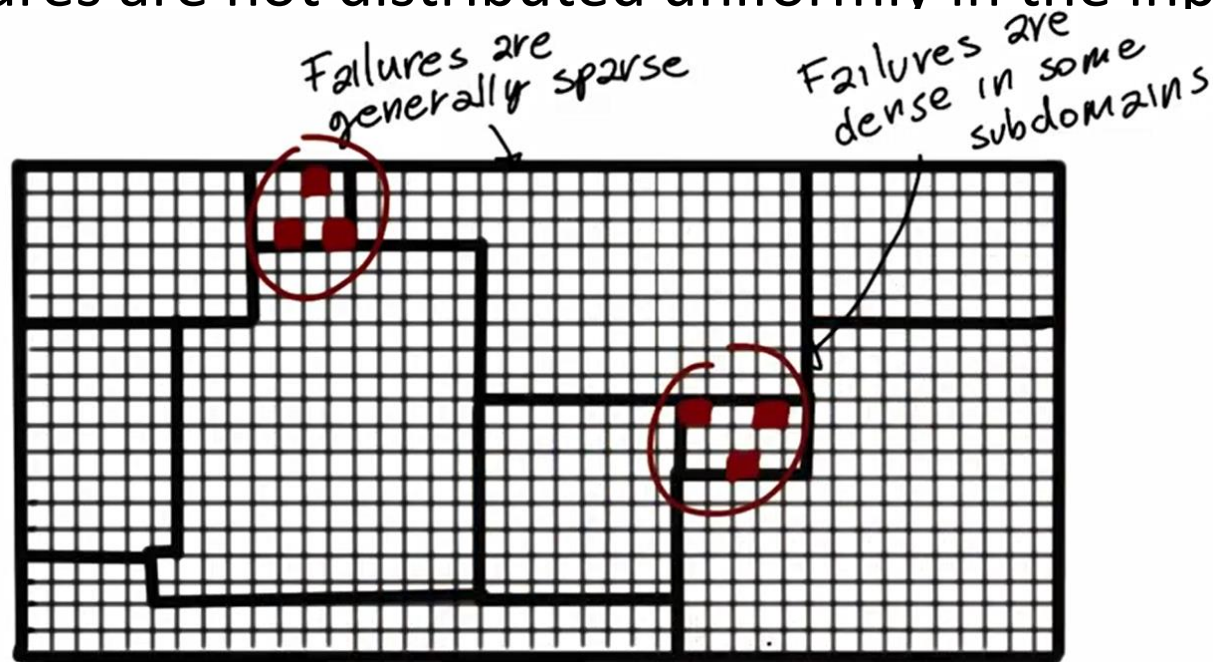
```
1. int sum(int a, int b) {  
2.     return a + b;  
3. }
```

*~ 600 years!*

- *How can we select interesting inputs?*

# Test Data Selection - partitioning

- Insight: failures are not distributed uniformly in the input domain!



pic source: Alex Orso, Software Development Process class notes

- Identify partitions and select inputs from each partition

# Example – partitioning

```
1. int Split(string str, int size) {  
2.     //split the input string str into chunks of  
     //length size  
3. }
```

- Some possible partitions:
  - `str` with length `< size`
  - `str` with length `= [size, sizex2]`
  - `str` with length `> sizex2`
  - `size < 0`
  - `size > 0`
  - `size = 0`

Note that testing for `size < 0` overcomes developer bias

# Example – partitioning

- All inputs in a sub-domain may not be interesting!
  - Errors tend to occur at **the boundary** of a sub-domain
  - why? scenarios not well understood by developers

- Some possible inputs (based on partitioning done earlier):

(consider boundary of size  $< 0$  i.e. size=-1, boundary of length of str  $<$  size i.e. length = size - 1)

– size = -1, length of str = size - 1 ←

– size = 1, length of str = size - 1

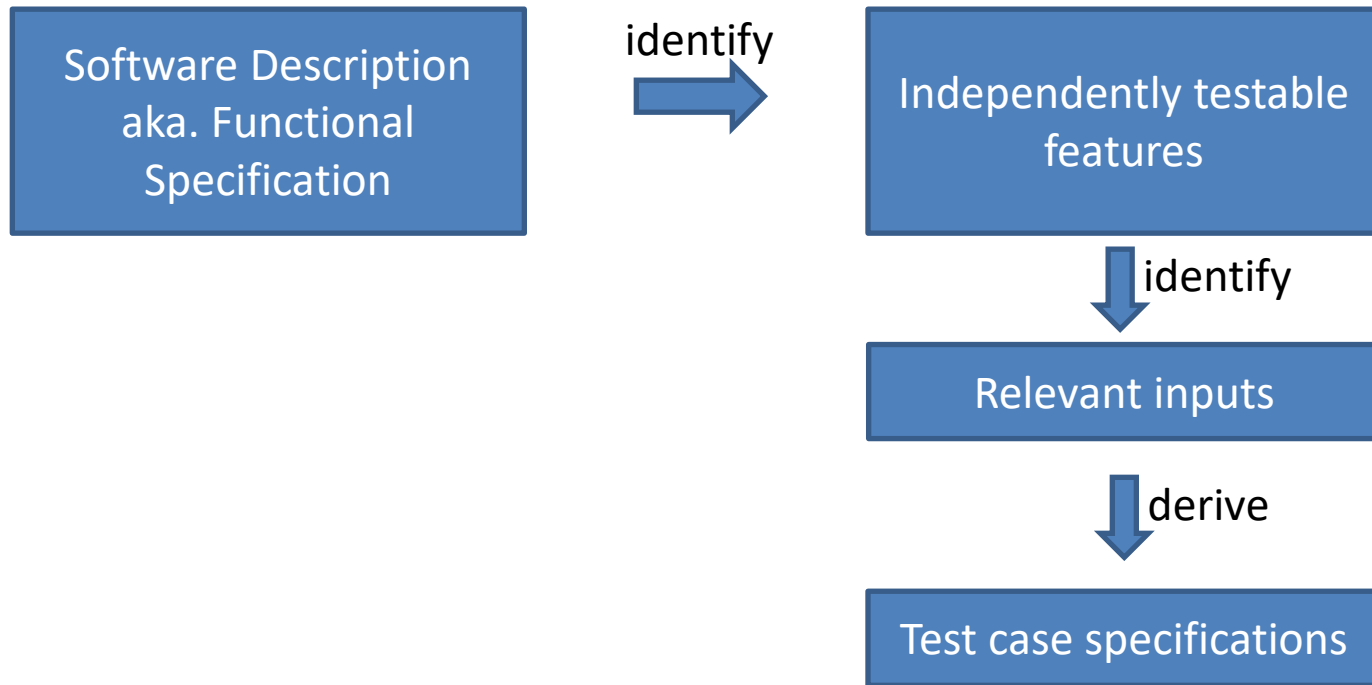
– size = -1, length of str = size (consider boundary of size  $> 0$  i.e. size=1, boundary

– size = 1, length of str = size ← of length of str = [size, size+1] i.e. length = size - 1)

**Key: adapt based on domain and type of data**

# From Spec. to Test Cases

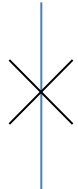
- **Step 3:** derive test case specifications





# Example – deriving test case specs

- Some possible inputs (based on partitioning done earlier):

size < 0,  str with length = size - 1  
size > 0, str with length = size  
size = 0

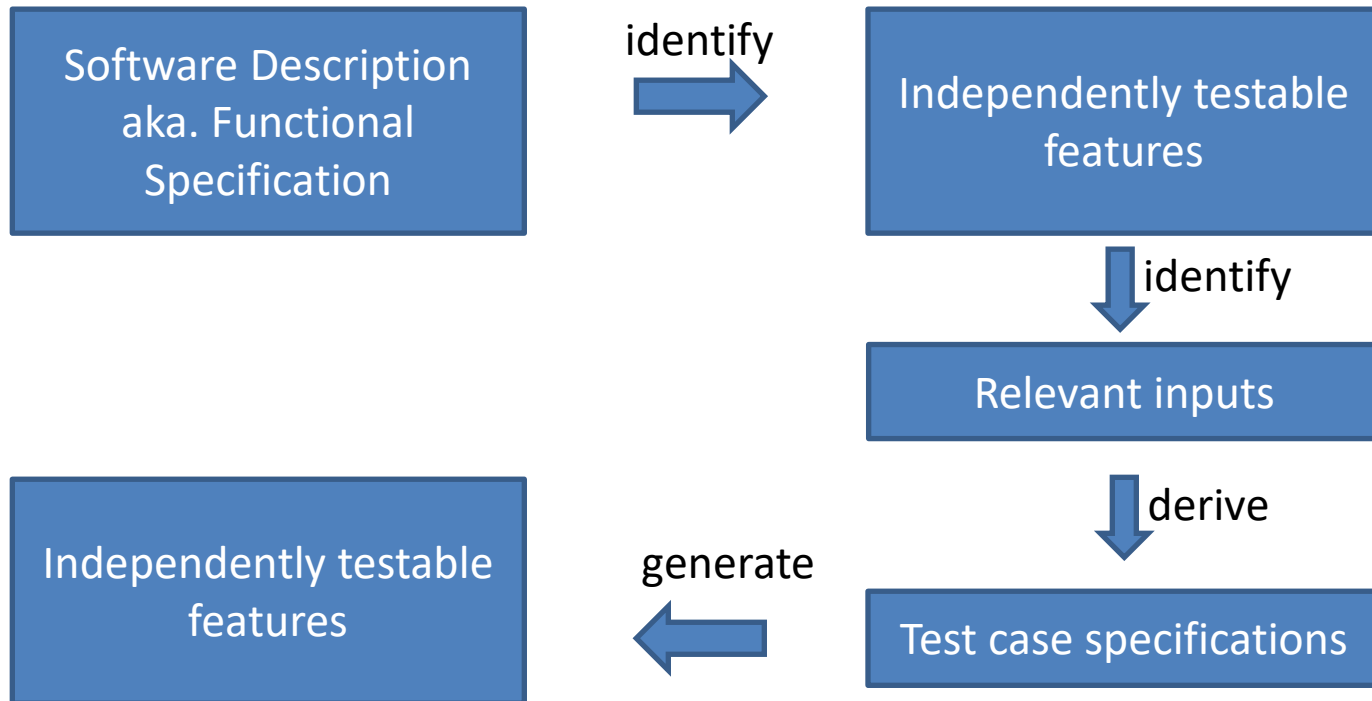
- Test case specifications:

- size = -1, length(str)=-2
  - size = 1, length(str)=0
  - size = -1, length(str)=-1
  - size = 1, length(str)=1
- Meaningless to have length(str) < 0  
Can prune / remove the test case.

- Limit the number of test cases generated

# From Spec. to Test Cases

- **Step 4:** generate test cases from specifications



# Example – test case generation

- Test case specifications:
  - `size = 1, length(str)=0`
  - `size = 1, length(str)=1`
  - ...
- Instantiate test cases from specs.
  - `Split("",1) //compare the result with expected output`
  - `Split("a", 1) //compare the result with expected output`

# From Spec. to Test Cases - Remarks

- The previously outlined approach is systematic:
  - Decoupling different activities
  - Separating analytical-intensive tasks from those that are not
  - Monitoring testing process e.g. not generating too many test cases

# Black-Box Testing – Category Partition Method

- Ostrand and Balcer defined it in 1988
- Defines method to create a set of test cases from specification

- 6 steps:

Seen previously in the general approach

1. Identify independently testable features ✓
2. Identify categories
3. Partition categories into choices
4. Identify constraints among choices ✓
5. Produce (and evaluate) test case specifications ✓
6. Generate test cases from test case specifications ✓

# Identify categories - Example

– `Split( string str, int size)`

input str:  
Has length

input size:  
Has value

Has some content

# Partition categories - Example

– Split( string str, int size)

input str:

Has length

- 0

- size - 1

. . .

Has some content

- has spaces

- has special characters

. . .

input size:

Has value

- 0

- > 0

- < 0

- MAXINT

. . .

Goal: identify interesting subdomains

# Identify Constraints - Example

- Use properties IF, ERROR, SINGLE
- Goal:
  - to eliminate meaningless combinations
  - to reduce the number of test cases

Split( string str, int size)

input str:

Has length

- 0 **PROPERTY ZEROVAL**

Has some content

- has spaces **consider IF !ZEROVAL**

input size:

Has value

< 0 **ERROR**

consider cases that combine “has spaces” with any other category *except those with property ZEROVAL*



# Identify Constraints - Example

Split( string str, int size)

input size:

Has value

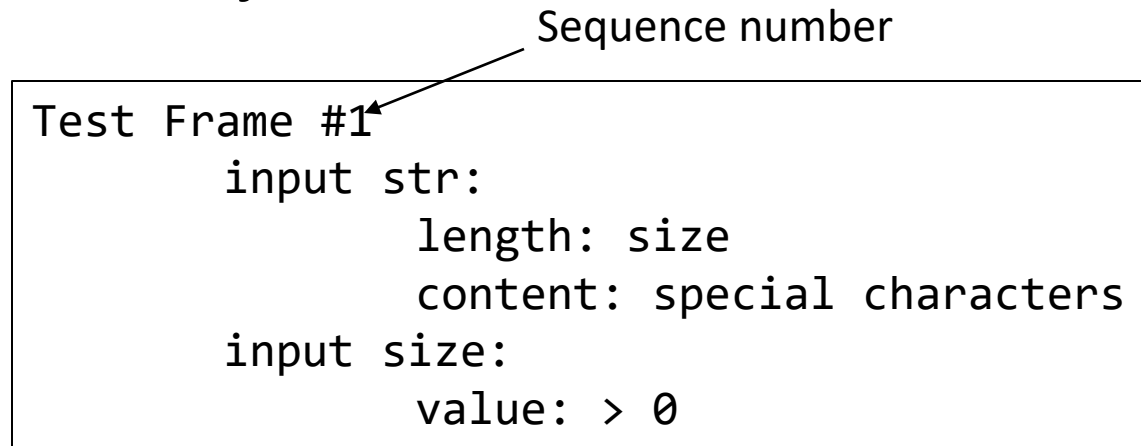
- < 0 **ERROR**
- = MAXINT **SINGLE**

consider cases that combine < 0 only once because we assume that we want to test error scenario (invalid parameter value passed) only once

Used to limit the number of test cases. Use this category only once in any combination of test cases.

# Produce and Evaluate Test Case Specs.

- Produces *test frames*

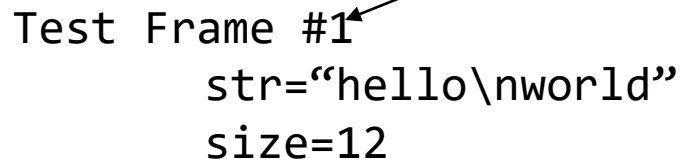


- Evaluate to see if you end up with too many test cases based on the combination of categories. If so, add more constraints.

# Produce Test Cases from Specs.

- Produces concrete *test cases*

Sequence number



```
Test Frame #1
    str="hello\nworld"
    size=12
```

# Test Specification Generator

- TSL: <https://github.com/alexorso/tslgenerator>
- Developed by professors from UC Irvine and U Oregon
- Implements Category-Partition method

Demo

# Specification

---

## NAME

**grep** - search a file for a pattern

## SYNOPSIS

**grep** <pattern> <filename>

## DESCRIPTION

The `grep` utility searches files for a pattern and prints all lines that contain that pattern on the standard output. A line that contains multiple occurrences of the pattern is printed only once.

The pattern is any sequence of characters. To include a blank in the pattern, the entire pattern must be enclosed in single quotes ('). To include a quote sign in the pattern, the quote sign must be escaped (\'). In general, it is safest to enclose the entire pattern in single quotes '...!.

# Output: Test Specs

```
Test Case 10          <single>
  Presence of quotes within the pattern : Many

Test Case 11          <error>
  Presence of a file corresponding to the name : Not present

Test Case 12          (Key = 2.2.1.3.3.1.1.1.2.)
  Size                : Not empty
  Number of occurrences of the pattern in the file : One
  Number of occurrences of the pattern in one line : One
  Position of the pattern in the file             : Any
  Length of the pattern                          : More than one
  Presence of enclosing quotes                   : Not enclosed
  Presence of blanks                             : None
  Presence of quotes within the pattern          : None
  Presence of a file corresponding to the name   : Present

Test Case 13          (Key = 2.2.1.3.3.1.1.2.2.)
```